Algorithms and Data Structures course overview

This is a quick seven-page summary of the contents of the algorithm and data structures course. We will begin by building up the necessary mathematical theory necessary to discuss applicable data structures and algorithms; we then proceed by looking at various data structures for storing, accessing and manipulating linearly ordered data, then data without relationships, and then graphs. This will be followed by a smaller section devoted to algorithms including algorithm design techniques and an introduction to the theory of computation.

Building the mathematical foundations

In software engineering, programming, and life in general, we need to store collections of numbers, values, or other objects. A class that implements a mechanism for storing objects is said to be a *container*. There may be things we want to do to objects in a container: add more objects, remove objects from the container, query the objects in the container, rearrange them, or possibly perform other operations such as combining the contents of two containers into one.

In addition to storing objects, we may also want to store relationships between those objects and then perform queries or other operations based on the relationships. This course looks at six relationship, including

linear, hierarchical, partial, and weak.

These are classified as *orderings* where if $x \prec y$, then $y \not\prec x$. In English, if *x comes before y*, it cannot be true that *y* also comes before *x*. The other two relationships are

equivalence and adjacency.

In the first case, objects are grouped into things that are *equal* or *equivalent* according to some definition. The second is a definition used in this class to describe more general relationships where one object may or may not be considered to be adjacent to another.

Now, based on storing objects and relationships between the objects, there are certain types of containers that are used over and over again throughout engineering. These *patterns* are given special names to identify them so that they need not be described each time they are used. Such patterns are generally described as *Abstract Data Types* or *ADTs*. The ADTs we will look at in this class include

1. lists and strings;
2. stacks, queues and deques;
3. sorted lists and priority queues; and
4. directed acyclic graphs and graphs.

The abstraction only describes the behaviour: a queue has a first-in—first-out character with respect to its two operations for inserting (pushing or enqueuing) into and removing (popping or dequeuing) from such a queue container.

The next questions is one of implementation: how do we actually design and implement an abstract container? What data structures do we use? Do we use node-based data structrues such as linked lists or trees? Do we use arrays? Do we use a combination of various data structures? We may have operations or queries we may want to perform and we may develop algorithms to implement those queries or operations. Thus, we must question how are we to compare implementations? Which implementation is better? Do different implementations have different strengths and weaknesses?

The criteria by which we analyze various implementations and algorithms could include such soft criteria such as expected development and maintenance costs; however, we will focus on those criteria that we can measure mathematically: run time and memory usage; that is, how fast can it run under certain circumstances (usually how fast are operations when a container already has n objects in it) and how much memory does it require (usually a container storing n objects will occupy some multiple of n bytes, but some queries and operations may require additional memory)?

Given any implementation or algorithm, we could find an exact mathematical function that describes exactly the run time and the memory usage; however, we will find that this is often unnecessary overkill. Instead, to practically compare the run-time and memory usages of various functions, we have less interest in those differences that can be practically solved by upgrading our hardware. Consequently, we consider the asymptotic behaviour between two function $f(n)$ and $g(n)$ by considering the limit $\lim_{n\to\infty}\frac{f(n)}{g(n)}$. There are three cases of interest for the types of functions we will consider:

1. The limit is zero: that is, proportionally, $f(n)$ grows significantly slower than $g(n)$.
2. The limit is finite but non-zero: that is, the growth of the functions is proportional to the other.
3. The limit is infinity, in which case, $f(n)$ grows proportionally more quickly than $g(n)$.

The mathematical tool that describes these three are Landau symbols. Respectively, we write: $f(n) = o(g(n))$, $f(n) = \Theta(g(n))$ and $f(n) = \omega(g(n))$. In the second case, it will always be theoretically possible to purchase proportionately faster or more memory to have one implementation or algorithm run as fast as another. (Note that while, in general, we will ignore such differences, there is one case, that of B+ trees, where we will take the proportion into account as the ratio is a matter of at least six orders of magnitude.) The balance of the course will comprise a larger section looking at data structures followed by one looking at algorithms, specifically, algorithm design techniques and an introduction to the theory of computation.

Data structures

Thus, we will begin by looking at ADTs storing linearly ordered data. We begin by looking at three specializations of abstract lists where there the interface is reasonably restricted: stacks, queues, and deques. The specifications for these is so simple that it is reasonable to expect all run times to be $\Theta(1)$. While considering these three, we looked at the asymptotic consequences of resizing arrays, the concept of amortized asymptotic analysis, the use of cyclic arrays, the use of queues in breadth-first traversals of trees, and the concept of an iterator.

We continue by considering abstract sorted lists. Arrays and linked lists are not appropriate underlying data structures for storing sorted lists because there are always some operations that must run in $\Theta(n)$ time. Instead, we take a diversion to trees.

A rooted tree data structure at first appears to be most appropriate for storing hierarchically ordered objects. There is a single root node and each node has a reference to its parent and its children. We considered one concrete implementation of a tree using a linked list of pointers to store references to the children of a node while storing a pointer to the parent in a separate member variable, this variable being assigned null if the node is the root of a tree.

Next, we consider visiting all the nodes in a tree. We have already seen a breadth-first traversal, but the memory requirements for such a tree may be prohibitively expensive, so we consider depth-first traversals.

Next, we look at ordered trees, including binary trees and N-ary trees with applications including expression trees and phylogenetic trees in the first case and tries in the second. We consider the special cases of perfect binary and N-ary trees to see that we can store n nodes in a tree of height $\ln(n)$. We also define complete binary and N-ary trees and observe that we can efficiently store such trees in an array where there is an implicit formula for calculating the location of the parent and the children of a node stored at a specific index.

With binary trees defined, we proceed to considering binary search trees: trees where all nodes in the left sub-tree are less than the current node and all nodes in the right sub-tree are greater than the current node. We observe that many operations are $O(h)$, but in general, $h = O(n)$, so if the distribution of nodes within a binary sub-tree is to *unbalanced*, we will end up with operations that run in $O(n)$ time—something that is no better than using arrays or linked lists.

Thus we introduce the concept of balance. The ideal case is something similar to either perfect or complete binary trees, but this is unreasonable—maintaining such a shape following either insertions or deletions would require $O(n)$ time. Instead, we consider relaxed definitions of balance including height, null-path, and weight balancing. We specifically look at AVL trees that use height balancing. We observe that the additional work required to maintain AVL balance for insertions is $\Theta(1)$ for insertions and we also showed that if a tree is AVL balanced, its height is, in the worst case, $\log_{\phi}(n) - 1.3277 = \Theta(\ln(n))$. Consequently, all operations that are $O(h)$ in a binary search tree must be $O(\ln(n))$ in an AVL tree.

Next, we consider in-order traversals and observe that while applying to binary trees, the do not make sense for N-ary trees and that N-ary trees with $N > 2$ cannot be used to store sorted lists. Instead, we consider the concept of an M-way tree. Like binary search trees, M-way trees have a height that is $\Omega(\ln(n))$. We cannot, however, use the rules of AVL trees to ensure the height of an M-way tree remains fixed at $\Theta(\ln(n))$. Instead, we consider B+ trees.

A B+ tree uses M-way nodes internally, and the additional complexity of performing operations within the nodes is observed to cancel any benefit from the reduced height except in one case: where the operation of stepping from a parent to a child is prohibitively expensive. We saw that this is the case if the nodes are being stored in secondary memory such as hard drives. We see that the restrictions to the distribution of values stored in internal and leaf nodes ensures that the height remains $\Theta(\ln(n))$ and that consequently, any operation that runs in $O(h)$ time will run in $O(\ln(n))$ time in a B+ tree.

We continue by looking at abstract priority queues. The definition is similar to that of an abstract queue, but the objects are stored relative to a linear ordering imposed by a priority assigned to each—thus, a priority queue is a specialization of an abstract sorted list. We could use an AVL tree to implement a priority queue, but all operations would run in $\Theta(\ln(n))$ time. Instead, we turn to a heap data structure and observe that with a complete binary heap, we can implement a priority queue where the *front* operation runs in $\Theta(1)$ time, push in an amortized $\Theta(\ln(n))$ time and pop in $O(\ln(n))$, but with an expected run time of $\Theta(1)$ given random priorities on incoming entries.

Next, we will look at sorting algorithms for converting a list into a sorted list. We see that all algorithms run in $\Omega(n)$ time, but that those algorithms that compare entries, they require $\Omega(n \ln(n))$ time on average. We will look at insertion and bubble sort which run in $\Theta(d)$ time where d is the number of inversions and where $d = O(n^2)$. We will also look at heap, merge, and quick sort, all of which run on average in $\Theta(n \ln(n))$ time, but where quick sort runs in $\Theta(n^2)$ time and where merge sort requires $\Theta(n)$ additional memory, quick sort requires $O(n)$ additional memory, but on average $\Theta(\ln(n))$ additional memory, and heap sort requires $\Theta(1)$ additional memory. Unfortunately, heap sort is also the slowest of the three of these, so we consider modifications to quick sort that minimize the probability of the worst case scenarios.

There is an optional reading topic on splay trees.

This concludes our investigations of storing linearly ordered data. We considered both abstract lists and abstract sorted lists and specializations of each. We also quickly investigated abstract trees for storing hierarchically ordered data. Next, we look at storing data without any relationship.

Anything stored on a computer must be stored as ones and zeros; consequently, it is possible to linearly order anything and therefore we could store anything we want in, for example, an AVL tree. However, there are times where we have no interest in that linear relationship. (For example, who cares whose student ID number the next largest of your own?) In these cases, we can use the concept of hash functions and hash tables to store data so that, on average, all operations of interest (inserting into, accessing, removing from a container) may be performed on average in $\Theta(1)$ time. We will look at chained hash tables and the open addressing implementations using linear probing and double hashing.

There is an optional reading topic on disjoint sets used to store equivalence relations on a finite number of objects.

Next we consider the implementation of graphs and directed acyclic graphs for storing data with adjacency relations and partial orderings, respectively. We will look at three problems: finding a topological sorting of directed acyclic graphs, finding minimum spanning trees within weighted graphs, and finding single-source shortest distances, also in weighted graphs. We will use a simple algorithm to find a topological sorting by observing that every directed acyclic graph has at least one vertex with in-degree zero. The run time will be $O(|V| + |E|)$. To find a minimum spanning tree, we will build a global minimum spanning tree by extending minimum spanning trees on sub-graphs, an algorithm attributed to Prim. Following this, we will look at the problem of finding the shortest distance from a given vertex to every other vertex in the graph. Specifically, we will consider Dijkstra's algorithm which has many characteristics similar to Prim's algorithm. In both cases, judicious choices made in how we store the graph and how we store intermediate information used in the algorithm will allow us to solve these problems in $O(|E| \ln(|V|))$ time.

This concludes our investigations into storing data with or without relationships in various data structures. We have seen a number of approaches to solving some of the problems, and we will continue by formalizing those approaches.

Algorithms

The next component considers algorithm design techniques. We look at greedy algorithms, divide-and-conquer algorithms including a proof of the master theorem, dynamic programming, backtracking algorithms, branch and bound algorithms, and finally stochastic algorithms.

Greedy algorithms are those where a partial solution is built up from a trivial partial solution where, at each step, a simple decision is made as to how to extend the partial solution. Such a sequence of partial solutions may grow into either an optimal or possibly a near optimal feasible solution. It is when the problem has an optimal sub-structure characteristic, that is, the partial solutions are known to be optimal for that sub-graph on which it is defined, that the global solution is also known to be optimal. We will look at a project management example (the 0/1 knapsack problem), process scheduling, and interval scheduling.

Divide-and-conquer algorithms take a larger problem and devises sub-problems that are smaller in size to solve. These sub-problems are solved recursively and the solutions to the sub-problems are then combined to generate a solution to the larger problems. In general, a problem is divided into a sub-problems of size n/b and the effort required to generate the sub-problems together with the effort required to generate a solution from the sub-problems is considered to be polynomial of the form $O(n^k)$. We will consider numerous applications including merge sort, searching an ordered matrix, integer multiplication, matrix-matrix multiplication, and the fast Fourier transform. Having considered these examples, we will then deduce a general formula given by the *master theorem* which describes the run-time behavior of all such divide-and-conquer algorithms.

Next we consider dynamic programming. We will look at two simple recursive algorithms, namely calculating the factorial and Newton polynomial coefficients. To discuss the approaches, we will define both top-down design and bottom-up design of algorithms. In both cases, the naïve top-down implementation will be seen to run in exponential time. We will see how a bottom-up design can be used to build solutions from previous solutions, but at the same time, we will see how a simple application of memoization will result in significant speed ups for top-down solutions. We will then look at optimization problems, where the recursive algorithms are repeatedly seeking optimal solutions on sub-problems. It is when optimal solutions are sought multiple times on the same sub-problem (*overlapping sub-problems*) that memoization helps significantly. We will look at matrix chain multiplications with both top-down and bottom-up designs and next we will see how optimal polygon triangulation is, while initially seeming to be a very different algorithm, is in fact a very similar problem, and finding a solution in one allows one to find a solution to the other. This introduces a concept of reduction which will be expanded upon later. We will then look at interval scheduling and a restricted version of the 0/1-knapsack problem. An overview of the Maple programming language's *remember tables* will demonstrate how memoization can be built into a programming language.

Next we will look at backtracking algorithms for a mechanism to systematically search through a space of possible partial solutions by using an appropriate traversal. In some cases, it will be possible to deem unfeasible entirely sub-branches of the traversed tree thereby reducing the search from what would normally be equivalent to a brute force search. We look at an implementation of an algorithm to solve the Sudoku puzzle and we look at other class puzzles in this area. We will also look at how backtracking is used in logic programming language such as Prolog and we will consider parsing with respect to context-

free grammars. Finally, we consider the idea of *backjumping*, looking specifically at algorithms for finding solutions in mazes.

Next we will consider branch-and-bound algorithms, algorithms similar to backtracking but where, when looking for optimal solutions, under certain conditions we can place bounds on the best possible solution at various points in the tree. If a particular branch is known to be worse than the best currently known solution, we can *prune* that branch and investigate elsewhere. We look at the game of Backgammon as an example where branch and bound may be used to choose reasonably optimal solutions in a stochastic environment.

Finally, we consider stochastic algorithms: we will consider random number generation techniques, Monte Carlo techniques for approximating integrals, testing of circuit designs for stability in light of imperfect components, a randomized variation of quicksort that prevents the deliberate insertion of worst-case scenarios, and skip lists.

Having finished with algorithm design techniques, we conclude by having an overview of the theory of computation. We will discuss models of computation and define the Turing machine as a model of what can be computed. We will then observe that there are problems that cannot be solved; specifically we will see that it is not possible, in general, to find an algorithm that can determine if another algorithm, given a set of inputs, will go into an infinite loop. We will then classify problems into how tractable they are to solve. Most of the problems we have looked at can be solved in polynomial time. There are problems, however, that cannot be solved in polynomial time; however, we will look at one branch of problems that can be solved in polynomial time assuming our algorithm is allowed to branch. There is a subset of these problems where we are not entirely certain as to whether or not there are polynomial-time algorithms that can solve them. We will call these *NP* and a subset of these (*NP* Complete problems) are such that solving any one of them in polynomial time without branching allows you to solve all *NP* problems in polynomial time without branching.

Summary

Finally, we will conclude with a look at the old Yale sparse matrix format; a useful data structure that combines a lot of the ideas in this course and and ties the course material to the linear algebra course.